Partical Swarm Optimization

Application : Traveling sales Man

```
import numpy as np

# Function to calculate the total distance of a route (path)
def calculate_total_distance(route, distance_matrix):
    total_distance = 0
    for i in range(len(route) - 1):
        total_distance += distance_matrix[route[i], route[i + 1]]
    total_distance += distance_matrix[route[-1], route[0]]  # Return to start
    return total_distance

# Particle Swarm Optimization (PSO) for TSP
class PSO_TSP:
    def __init__(self, distance_matrix, num_particles=30, num_iterations=100, w=0.5, c1=1, c2=1):
        self.num_particles = num_particles
        self.num_iterations = num_iterations
        self.distance_matrix = distance_matrix
        self.num_cities = len(distance_matrix)
        self.w = w  # Inertia weight
        self.c1 = c1  # Cognitive coefficient
        self.c2 = c2  # Social coefficient

        # Initialize particles' positions (routes) and velocities
        self.particles = np.array([np.random.permutation(self.num_cities) for _ in range(num_particles)])
        self.velocities = np.array([np.zeros(self.num_cities) for _ in range(num_particles)])

        # Evaluate fitness of each particle (route)
        self.fitness = np.array([calculate_total_distance(route, distance_matrix) for route in self.particles])

        # Initialize personal best positions and fitness
        self.p_best = np.copy(self.particles)
        self.p_best_fitness = np.copy(self.fitness)
```

```
        # Initialize global best position and fitness
        self.g_best = self.p_best[np.argmin(self.p_best_fitness)]
        self.g_best_fitness = np.min(self.p_best_fitness)

    # Update velocities and positions
    def update_particles(self):
        for i in range(self.num_particles):
            # Update velocity: w * velocity + c1 * random() * (personal best - current position) + c2
* random() * (global best - current position)
            r1 = np.random.rand(self.num_cities)
            r2 = np.random.rand(self.num_cities)
            cognitive_velocity = self.c1 * r1 * (self.p_best[i] - self.particles[i])
            social_velocity = self.c2 * r2 * (self.g_best - self.particles[i])
            inertia_velocity = self.w * self.velocities[i]
            self.velocities[i] = inertia_velocity + cognitive_velocity + social_velocity

            # To ensure we move to a new route, modify the velocity to shuffle positions
            velocity_order = np.argsort(self.velocities[i])  # Sort based on the velocity magnitude
            new_particle = np.array([self.particles[i][j] for j in velocity_order])

            # Ensure the new particle is a valid permutation
            self.particles[i] = new_particle
            self.fitness[i] = calculate_total_distance(new_particle, self.distance_matrix)

            # Update personal best
            if self.fitness[i] < self.p_best_fitness[i]:
                self.p_best[i] = self.particles[i]
                self.p_best_fitness[i] = self.fitness[i]

            # Update global best
            if self.fitness[i] < self.g_best_fitness:
                self.g_best = self.particles[i]
                self.g_best_fitness = self.fitness[i]

    # Run the PSO algorithm
    def run(self):
        for iteration in range(self.num_iterations):
            self.update_particles()
```

```
            print(f"Iteration {iteration + 1}: Best Distance = {self.g_best_fitness}")
        return self.g_best, self.g_best_fitness

# Function to take user input for distance matrix and PSO parameters
def input_pso_parameters():
    # Input the number of cities and distance matrix
    num_cities = int(input("Enter the number of cities: "))
    print("Enter the distance matrix row by row (space-separated):")
    distance_matrix = np.zeros((num_cities, num_cities))
    for i in range(num_cities):
        row = list(map(int, input(f"Row {i + 1}: ").split()))
        distance_matrix[i] = row

    # Input PSO parameters
    num_particles = int(input("Enter the number of particles: "))
    num_iterations = int(input("Enter the number of iterations: "))
    w = float(input("Enter the inertia weight (w): "))
    c1 = float(input("Enter the cognitive coefficient (c1): "))
    c2 = float(input("Enter the social coefficient (c2): "))

    return distance_matrix, num_particles, num_iterations, w, c1, c2

# Get user input for the distance matrix and PSO parameters
distance_matrix, num_particles, num_iterations, w, c1, c2 = input_pso_parameters()

# Initialize PSO with the distance matrix and parameters
pso_tsp = PSO_TSP(distance_matrix, num_particles, num_iterations, w, c1, c2)

# Run PSO to find the shortest path
best_route, best_distance = pso_tsp.run()

print("\nBest route found:", best_route)
print("Best route distance:", best_distance)
```

Output :

```
Enter the number of cities: 4
Enter the distance matrix row by row (space-separated
Row 1: 0 5 10 15
Row 2: 5 0 20 30
Row 3: 30 10 0 5
Row 4: 5 10 15 0
Enter the number of particles: 50
Enter the number of iterations: 200
Enter the inertia weight (w): 0.7
Enter the cognitive coefficient (c1): 1.5
Enter the social coefficient (c2): 1.5
Iteration 1: Best Distance = 30.0
Iteration 2: Best Distance = 30.0
Iteration 3: Best Distance = 30.0
Iteration 4: Best Distance = 30.0
Iteration 5: Best Distance = 30.0
Iteration 6: Best Distance = 30.0
Iteration 7: Best Distance = 30.0
Iteration 8: Best Distance = 30.0
```

```
Iteration 185: Best Distance = 30.0
Iteration 186: Best Distance = 30.0
Iteration 187: Best Distance = 30.0
Iteration 188: Best Distance = 30.0
Iteration 189: Best Distance = 30.0
Iteration 190: Best Distance = 30.0
Iteration 191: Best Distance = 30.0
Iteration 192: Best Distance = 30.0
Iteration 193: Best Distance = 30.0
Iteration 194: Best Distance = 30.0
Iteration 195: Best Distance = 30.0
Iteration 196: Best Distance = 30.0
Iteration 197: Best Distance = 30.0
Iteration 198: Best Distance = 30.0
Iteration 199: Best Distance = 30.0
Iteration 200: Best Distance = 30.0

Best route found: [2 3 1 0]
Best route distance: 30.0
```